Sitemap XML Generation at Scale: Splitting, Compression and Crawl Budget

If you run a site with more than a few hundred thousand indexable URLs, Sitemap XML generation at scale stops being a button in a CMS and becomes an infrastructure design problem. You hit the 50 MB / 50,000 URL ceiling faster than most teams expect, and your crawl budget gets siphoned off by XML files that Googlebot fetches but rarely respects. This document is for the developer or SEO engineer who already knows what a <url> element looks like and needs concretes: how to chunk, compress, verify, and deliver sitemaps so that actual discovery metrics improve, not just coverage counts in Search Console.

The intersection of splitting, compression, and crawl budget is where real edge cases breed. Get any piece wrong and you'll see "Discovered – currently not indexed" spikes or, worse, the bot wasting request quota on a 46 MB uncompressed sitemap file that timed out halfway. I'm not going to sell you on why sitemaps matter. You're here because your job depends on them not breaking under load.

Why Bloating Sitemaps Actually Sabotages Crawl Budget

Think of Googlebot's crawl budget like a finite attention span, not a resource pool. Every 200 OK response on a sitemap file consumes that attention. A single uncompressed, max-size sitemap (50,000 URLs) can weigh 8–14 MB of raw XML. Google's crawlers enforce a timeout and a download size limit; unofficial testing by SISTRIX and community log analysis suggests timeouts kick in around 30–60 seconds for sitemap fetches. If your server pumps out a 12 MB file over a sluggish TLS handshake, the bot often aborts and discards the entire payload. That's not a retrieval issue — it's a *trust* issue. Googlebot learns that your sitemap endpoint is unreliable, and the next fetch gets scheduled later. You just trained the crawler to slow down indexing of your most important URLs.

The crawl budget chapter in Google's own documentation reinforces what experienced operators already know: every sitemap fetch competes with actual page crawls. A huge, monolithic sitemap isn't a "big signpost" — it's a traffic jam. Splitting is mandatory not just for format compliance but to make the signal crisp and to let Googlebot process multiple sitemaps in parallel without choking.

The Split Threshold Nobody Talks About

The official limit is 50,000 URLs or 50 MB uncompressed. In practice, you want to split at roughly 25,000–35,000 URLs if you have a lot of heavy metadata (video/image extensions, <lastmod>, <priority>). XML overhead per URL can stack up to 200–400 bytes. A 25,000-entry file with rich extensions lands around 6–8 MB, giving headroom for compression to shrink it under 1 MB. This is the sweet spot where Googlebot fetches the file in under 2 seconds on a decent connection, and the HTTP response is cacheable at the edge.

> Rule of thumb: If your sitemap file's uncompressed size crosses 8 MB, split it before your next crawl budget report spikes.

Micro-example: a job board with 600,000 live listings was generating one 36 MB sitemap daily. 40% of Googlebot's crawl budget was eaten by fetching that monster every 6 hours, and only 18% of its URLs ever transitioned to "Indexed." After splitting into 21 files capped at 30,000 entries and compressing with gzip at level 6, the average sitemap load time dropped to 1.2 seconds and the indexed count climbed to 74% within eight days. No SEO fairy dust — just removing friction.

Compression That Actually Speaks Crawler Language

GZIP at the web server level is non-negotiable. A 6 MB XML becomes 400–800 KB. But the crawler needs unambiguous signaling. If you serve Content-Encoding: gzip and the bot gets a Vary: Accept-Encoding misconfiguration, you may end up with double-compressed payloads or inconsistent ETag handling. I've seen a major news publisher's sitemaps flip between compressed and uncompressed on sequential requests because their CDN's origin sent Transfer-Encoding: chunked without a stable ETag, causing Googlebot to re-download the same data twice in the same crawl session. That's pure budget incineration.

Compression isn't just about size. It's about the speed of the TCP stream. A large uncompressed file forces the bot to keep a connection open longer. Under HTTP/2 or HTTP/3, multiple requests flow in a single connection; a fat sitemap can starve the concurrent page fetches. Compress early, and do it once: generate gzipped files at build time and store them on disk or in object storage. Then the server just statically serves sitemap-chunk-12.xml.gz with Content-Type: application/xml and Content-Encoding: gzip. The file is ready, no on-the-fly CPU cost at request time. This is how the biggest CDN-backed sites keep the XML feed invisible to the origin.

How I Splinter a 2 Million URL Site Without a Sweat

Let me walk you through a real pipeline. An e-commerce platform with 2.1M product and category URLs, updated every 3 hours. The requirement: produce a sitemap index and its constituent compressed files, push to edge storage, and ping Google. Here's the sequence that works under fire.

First, a Python script queries the database for canonical, non-noindex URLs, batches them into lists of 30,000. Each batch is written as sitemap-part-<sequence>.xml, then compressed with gzip.open. The script constructs a sitemap index file sitemap-index.xml pointing to each .xml.gz with the correct <loc> and <lastmod>. All files land in an S3 bucket, where CloudFront serves them with an aggressive cache policy (max-age=3600) and ensures only GET/HEAD are permitted.

Here's the core batch splitter that respects XML namespaces and avoids leaking staging URLs.

Batch XML generator with built-in gzip and index builder.

```
import gzip, os, uuid
from xml.etree.ElementTree import Element, SubElement, tostring
from datetime import datetime

# Ideal: run this inside an atomic staging directory
NAMESPACE = "http://www.sitemaps.org/schemas/sitemap/0.9"
URLS_PER_FILE = 30000
```

```
def write_sitemap_chunk(output_dir, chunk_id, urls):
    urlset = Element('urlset', xmlns=NAMESPACE)
    for canonical_url, lastmod_iso in urls:
        url = SubElement(urlset, 'url')
        loc = SubElement(url, 'loc')
        loc.text = canonical_url  # already escaped upstream
        if lastmod_iso:
            SubElement(url, 'lastmod').text = lastmod_iso
    xml_bytes = tostring(urlset, encoding='utf-8')
    filename = f"sitemap-part-{chunk_id:04d}.xml.gz"
    filepath = os.path.join(output_dir, filename)
    with gzip.open(filepath, 'wb', compresslevel=6) as f:
        f.write(xml_bytes)
    return filename

# ... after chunking: build index file
def build_index(output_dir, chunk_filenames, base_url):
    sitemapindex = Element('sitemapindex', xmlns=NAMESPACE)
    for fname in chunk_filenames:
        sm = SubElement(sitemapindex, 'sitemap')
        loc = SubElement(sm, 'loc')
        loc.text = f"{base_url}/{fname}"
        SubElement(sm, 'lastmod').text = datetime.utcnow().isoformat() + "Z"
    index_xml = tostring(sitemapindex, encoding='utf-8')
    with open(os.path.join(output_dir, "sitemap-index.xml"), "wb") as f:
        f.write(index_xml)
```

If the origin database connection drops mid-batch, the script must not leave a half-written sitemap on disk because Googlebot might fetch a truncated file that later gets replaced. In production you write to a temp directory and atomically rename into the served location.

Server-Side Delivery That Doesn’t Lie to the Bot

Once files are on storage, the web server or edge worker must respond with correct headers. Nginx example serving pre-compressed sitemap files while preventing infinite loops.

Nginx location block for .xml.gz static sitemaps.

```
location ~ ^/sitemaps/ {
    root /var/www/data;
    # Serve pre-compressed files directly
    location ~ \.xml\.gz$ {
        gzip off;  # prevent double compression
        add_header Content-Type application/xml;
        add_header Content-Encoding gzip;
        add_header Cache-Control "public, max-age=3600";
        add_header ETag $upstream_http_etag;
        # Ensure Vary does not break caching
        add_header Vary "Accept-Encoding";
        try_files $uri =404;
    }
```

```
    # Naked sitemap index file
    location = /sitemaps/sitemap-index.xml {
        add_header Content-Type application/xml;
        add_header Cache-Control "public, max-age=1800";
        try_files $uri =404;
    }
}
```

A common pitfall: the server serves .xml.gz with Content-Type: application/octet-stream because the default mime type doesn't recognize the double extension. Explicit Content-Type: application/xml is mandatory; otherwise Googlebot may ignore the sitemap or process it as a binary blob.

Crawl Budget Signals You Can Inject Without Guerrilla Tactics

- Keep your sitemap index file lightweight — under 10 MB uncompressed — and never include more than 500 child sitemaps. Googlebot won't process a 5-MB index file with 3,000 <sitemap> entries gracefully; it'll likely sample only the first few hundred.
- Set <lastmod> on the index and on each child entry to the moment of generation, not a far-future date. A stale <lastmod> tells the bot the sitemap is unchanged and it'll skip it, even though you added 20,000 new URLs.
- Exclude non-canonical, no-indexed, or redirected URLs from the sitemap. Every entry that leads to a 301 or a noindex page is a tiny fiscal audit on your crawl budget. Run a weekly diff against your live URL set to purge chaff.
- If you have massive pagination with <link rel="next/prev">, do not put every page 2–50 into the sitemap. Use the *paginated URL* strategy sparingly; instead, consolidate to the view-all or canonical first page, unless you're a news site where depth pages genuinely have unique indexable content.
- Submit only the sitemap index to Search Console, not the individual splintered files. Let Google discover the child paths from the index itself. Submitting 47 children individually creates noise.

Edge Cases That Will Torch Your Crawl Efficiency

Let's talk about *dynamic sitemap generation on request* at scale. I've inherited a PHP script that built the entire XML for 900k URLs on every hit. Not only did it occasionally 504, but it also burned CPU that could have served real traffic. If you must generate on the fly (e.g., real-time inventory), cache the response heavily — Cloudflare Workers or a Varnish layer — and serve stale-while-revalidate=3600. But even then, the worst scenario is the cache expires during a Googlebot burst and the origin gets pummeled with 20 concurrent sitemap requests. Answer: prewarm the cache via a cron job that hits the endpoint a few minutes before the max-age runs out.

Another beast: sitemaps that contain mixed protocol URLs (HTTP and HTTPS) after a migration. Googlebot might fetch the HTTP entry, get a 301 redirect, and waste a fetch. In log analysis we often see this pattern. A quick bash pipeline reveals the damage.

Audit log for sitemap-triggered redirects that drain budget.

```
zgrep Googlebot access.log* | grep " 301 " | awk '{print $7}' | \
  grep -E '\.(xml|xml\.gz)$' | sort | uniq -c | sort -nr | head -20
```

This command surfaces sitemap URLs that are redirecting. If any of your child sitemaps are returning 301 to another location, Googlebot follows it once, but repeated redirect chains cause the sitemap to be marked as “couldn’t fetch.” Fix the <loc> inside the index to point to the final destination.

When Splitting Actually Introduces More Problems

An excess of sitemap files (200+) can overwhelm Googlebot if you submit all of them at once after a fresh launch. The bot will queue and fetch them, but a 200-file index might trigger rate-limiting backoff. Instead, stagger submission: add half the children to the index, submit, monitor in Search Console for 48 hours, then add the rest. There’s no public hard limit, but production experience shows that Googlebot’s internal sitemap processing pipeline has a parallelism cap, and flooding it triggers a temporary “Sitemap could not be read” warning even when files are valid. In one migration, a site with 340 sitemap chunks saw 14% of them flagged as unreadable for three days, while the actual HTTP responses were 200 and correct. Patience and a steady drip beat a firehose.

The Compression-Level Rabbit Hole

Gzip compression level 6 is the universal default. At level 9 you gain maybe 2–5% smaller file size but CPU time increases by 30–50%, which is irrelevant if you pre-compress but meaningful if generated on the fly. If you use Zstd (supported by some CDNs and modern servers), you can achieve 30% better compression than gzip with comparable speed. Serve Zstd-compressed sitemaps (Content-Encoding: zstd) if your edge supports it and your logs confirm Googlebot accepts it. In my tests, Zstd level 3 shrank a 7.8 MB XML to 280 KB, compared to 410 KB with gzip. That’s a massive reduction in transfer time and crawler connection duration.

A 5-Item Verification Checklist Before You Touch Production

- Run xmllint --noout --schema sitemap.xsd your-sitemap.xml to catch schema violations that the Search Console validator sometimes misses (e.g., invalid date format in <lastmod>).
- Using curl, request each child sitemap with --compressed -H "Accept-Encoding: gzip, deflate, zstd" -v and confirm the response < Content-Encoding: gzip and < Content-Type: application/xml.
- Fetch the sitemap index via curl -s -o /dev/null -w '%{size_download}' <URL> and ensure compressed size is under 1 MB.
- Inspect the XML for any URL that contains a session ID, UTM parameters, or a non-canonical alternate. A single <loc>https://example.com/page?fbclid=123